N3519

Understanding Effective Type Aliasing in C

by Eskil Steenberg

The C programming language has since C99 a concept known as Effective type. The Effective type rules are designed to improve a compiler's ability to do aliasing analysis. These rules and their implications are understood by almost no one. By "almost no one" I mean probably less than ten people in the world. I have programmed C for 25 years, I have never even heard about it until recently. I represent Sweden in the ISO WG14 standard group, I'm in the C memory model study group, and only very recently did I grasp the concept. Several people in the WG14 has asked for a document to try to help them understand the concept of Effective type so I will now do my best to try to explain, how I think it works.

In this document I will try to explain what the standard says, not what implementations actually do. As far as I know, no compilers will break these rules, so if you write within these rules you should be fine (baring compiler bugs and there are a fair bit of them in this area given how few people understand this).

This is my best interpretation of the C standard. I have spent considerable time and effort in collaboration to try to understand these rules and their implications. I owe huge gratitude for the time and efforts of my peers helping me decipher this, especially Jens Gustedt and Martin Uecker. Still, this is only my interpretation, and it is not a document officially endorsed by the WG14 or the memory model study group.

Just one more note before we get started. This is a document trying to explain how the effective type system in C works, it is not an endorsement of its design.

A Primer on Aliasing.

Before we talk specifically about the effective type system we need to talk about what problem it is meant to solve.

Consider the following code:

```
typedef struct {
    unsigned int length;
    float *array;
} MyStruct;

void function(MyStruct *s)
{
    unsigned int i;
    for(i = 0; i < s->length; i++)
        s->array[i] = 0.0f;
}
```

How may a compiler go about optimizing this code? There are many different opportunities here, it could compute an end pointer and step forward towards it, it may make a call to an intrinsic memset, use vector instruction and all kinds of hardware-specific tricks, and simply putting the length value in a register. All these approaches depends on one thing: that writing to array does not overwrite the length member in struct. If the array pointer points to the length member, then this code means something very different from if it does not. If the compiler somehow is allowed to eliminate the possibility that writing to what array points to is an alias for writing to length.

This is known as aliasing. When something is being accessed by a pointer, and the same time being accessed directly or by another pointer, then the two alias. Consider this very simple example:

```
void function(float *a, int *b)
{
  *a = 2.0f;
  *b = 3;
}
```

If a and b are guaranteed to not alias, then the order of the two operations do not matter and they can even happen in parallel. If they do alias, the order of operation must be guaranteed.

Determining what aliases is therefore of great importance. Around the creation of C99 this had become an apparent problem, and one of the solutions was to add a set of rules, where the compiler was allowed to assume things about how memory was used, and any use of memory that breaks these rules us undefined behavior. Enter the “effective type” rules.

One way of determining if two values alias is to look at their types. This is know as type-based aliasing. C compilers employ type-based aliasing.

If we again consider the first example, one might think that since the “length” variable is of type unsigned int, and the “array” pointer points to a value of type float, it can be assumed that “array” doesn’t point to length. This assumption is wrong. C uses a system that depends neither on the type of the pointer nor the type of the value its points to, it depends on something entirely different: the Effective Type.

Most of the time, objects are only accessed using one type, and in these cases everything works as expected, but as soon as you want to access memory using more than one type, these rules kick in. We may for instance allocate memory that we want to re-use, we may cast pointers, we may want to do type punning, we may want to run encryption operation on arbitrary memory, or we may want to move memory using larger types rather than individual bytes.

The Rules

Let's start by looking at the rules themselves:

> The effective type of an object for an access to its stored value is the declared type of the object, if any. (Allocated objects have no declared type.) If a value is stored into an object having no declared type through an lvalue having a type that is not a character type, then the type of the lvalue becomes the effective type of the object for that access and for subsequent accesses that do not modify the stored value. If a value is copied into an object having no declared type using memcpy or memmove, or is copied as an array of character type, then the effective type of the modified object for that access and for subsequent accesses that do not modify the value is the effective type of the object from which the value is copied, if it has one. For all other accesses to an object having no declared type, the effective type of the object is simply the type of the lvalue used for the access.
>
> 7 An object shall have its stored value accessed only by an lvalue expression that has one of the following types: (The intent of this list is to specify those circumstances in which an object may or may not be aliased.)

- a type compatible with the effective type of the object,
- a qualified version of a type compatible with the effective type of the object,
- a type that is the signed or unsigned type corresponding to the effective type of the object,
- a type that is the signed or unsigned type corresponding to a qualified version of the effective type of the object,
- an aggregate or union type that includes one of the aforementioned types among its members (including, recursively, a member of a subaggregate or contained union), or
- a character type.

(Note: the term "Access" is defined as reading or modifying a value according to the C standard. This is a bit confusing since in one part of this text it says that you can sometimes change the effective type by overwriting a different type, and later it says you can only "access" (switching to) using a compatible type.)

What this boils down to is that if something has an effective type, you can only access it using a type compatible with that type. In most cases the effective type of an object is the same as the type. However sometimes this is not true, and we may want to access memory using many different types.

An Abstract Effective Type-based Architecture.

One way to think of it is to imagine a hardware architecture where next to every byte of memory there is a storage facility that stores the data type stored in that

byte. This facility is entirely separate from what type a pointer to the memory may have. Every time you write to a byte, the type storage associated with that byte is updated with the type you wrote with (with some exceptions), Every time you read from memory, you have to read using the same type as the type stored for that memory. If you don't, this imaginary architecture would fail. memcpy, memmove and byte copy are implemented as intrinsic and copy not just the memory, but also the type associated with the memory. The stack and variables with static storage duration automatically initialized types associated with all variables and the type data is write protected.

Examples:

(Note: For brevity this paper assumes that the types float and int have the same size, and that malloc and calloc don't return NULL.)

If you declare an int, it has the effective type of int and you have to use it as an int. So the following is undefined behaviour:

```
int i = 0;
float *fp;
fp = &i;
x = *fp;
```

i has an effective type of "int" because it was declared as an int and therefor it is UB to access it by dereferencing it as a float pointer. So far things are pretty straightforward. But what about allocated memory? Allocated memory is not declared, it's returned by functions like malloc, calloc and realloc. Consider this:

```
int *ip;
float *fp;
ip = calloc(sizeof *fp);
fp = (float *)ip;
x = *fp;
```

In this case the memory returned from calloc doesn't have an effective type, even if it's stored in to a pointer of type int. When memory doesn't have an effective type, reading using a type, gives the memory that effective type, so its entirely valid to also read it as a float. In fact we can read it as an integer too:

```
int *ip;
float *fp;
ip = calloc(sizeof *fp);
fp = (float *)ip;
y = *ip;
```

What we cannot do is read it as both:

```
int *ip;
float *fp;
ip = calloc(sizeof *fp);
```

```
fp = (float *)ip;
y = *ip;
x = *fp;
```

Here by first dereferencing it, as a int, we assign to the memory the effective type of int, and therefore dereferencing it as an float afterwards become UB.

Every time you write to an object with non-declared type, the effective type changes to the type you use to write to it. Consider:

```
int *ip;
float *fp;
ip = malloc(sizeof *fp);
*ip = 0;
fp = (float *)ip;
x = *fp;
```

This code is UB. Memory provided by malloc has no effective type, but as soon as we write to it using an int pointer, it gets assigned the effective type of int, and then any access to that memory has to be done using a type compatible with int. Given that float is not compatible with int, the dereference of fp becomes UB. Any write to allocated memory will automatically change the effective type, to the type being written (with an exception we will discuss later). So the following is not UB:

```
int *ip;
float *fp;
ip = malloc(sizeof *fp);
fp = (float *)ip;
*ip = 0;
*fp = 0.0f;
x = *fp;
```

In this case the memory returned by malloc has no effective type, but is then given the effective type of int as we write the integer 0 to it, then it's assigned the effective type float as we write the number 0.0 to the same memory, and then finally when we dereference a float pointer to the memory, that is legal since the memory has its effective type to float.

This tells us that allocated memory works fundamentally differently than declared memory:

- Declared memory has the same effective type as its declared type and can not change.
- Allocated memory can be given an effective type by writing to it or reading from it before it's given an effective type, and the effective type can change if one writes to it again using a different type.

The standard clearly says that the effective type of allocated memory changes when it is written to. For declared memory it doesn't, and it has to be a

compatible type. Consider:

```
float f;
int *ip;
ip = (int *)&f;
*ip = 42; // UB because we are accessing an incompatible type.
```

The Invisibility of Effective Type

A curious side effect of the effective type rules are that bugs that break the rules are often entirely invisible in the code where the bug happens. Consider the following:

```
void this_function()
{
    int *ip;
    ip = malloc(sizeof *ip);
    other_function(ip);
    printf("%i", *ip);
    free(ip);
}
```

This code can violate the effective type rules, and be UB depending on what other_function does with the allocated memory, even if the other_function is entirely free from UB. If the other_function writes to the memory using a type that in incompatible with int (something it is free to do) like this:

```
void other_function(void *p)
{
  *(float *)p = 3.14f;
}
```

...then the dereference of ip is UB. In isolation, neither this_function or other_function contains enough information to make the code UB. This problem does not only exist for the user, it also exists for the compiler. Consider this:

```
void function(int *ip, float *fp)
{
   ip[0] = 0;
   fp[0] = 1.0f;
   ip[1] = 2;
   fp[1] = 3.0f;
}
```

This function has two parameters, one float pointer and one int pointer. The types of the pointers say nothing about the effective types of the memory they point to. The effective type system allows for these to alias. Since the function never dereferences either of the pointers, the compiler can not use the effective type system to discern if the pointers alias or not. It is legal to overwrite the

same memory with different types as long as you read using the type compatible with your last write. This mean that a compiler can not use the effective type rules to say, optimize these four 32-bit writes in to two 64-bit store instructions.

A curious observation is that sometimes a compiler could assume that a particular memory must be allocated just by how it is accessed. Consider:

```
int function(float *fp)
{
  int *ip;
  float x;
  x = *fp;
  ip = (int *)fp;
  *ip = 42;
  return *ip;
}
```

Here, because the function uses the memory pointed to by fp to store and access two incompatible types, the memory must be allocated, otherwise this code is UB.

This means that technically, a function that takes a pointer to memory as one of its parameters may need to document in its usage requirements that the memory be allocated and not declared.

Let's return to the original example:

```
typedef struct {
    unsigned int length;
    float *array;
} MyStruct;

void function(MyStruct *s)
{
    unsigned int i;
    for(i = 0; i < s->length; i++)
        s->array[i] = 0.0f;
}
```

The naïve view is that a compiler can assume that "array" can't point to "length" because they are of different type, but this, as we have learned, is not true. In this case the compiler can assume that array doesn't point to length, because doing so would cause the memory to be assigned the effective type of float using the member "array", and then afterwards be read as an unsigned int by the "length" member, and this would be UB. The difference is subtle, but in this case the outcome is the same.

Exceptions

If we assume that all reads can only be done with the type something was written, then we quickly run in to a number of issues. Often in C you want to access and move memory in the form of arrays of bytes. The basic effective type rules would make this impossible. One such example is that it becomes impossible to implement or use memcpy. Therefore a number of exceptions have been carved out.

sign and qualifiers

The simple ones are integer sign and qualifiers. Sign and qualifiers are simply ignored when deciding if two types are compatible. A signed int is compatible with volatile unsigned int.

memcpy and memmove

memcpy and memmove have special properties that lets them write to both declared and allocated memory. However when they do write to allocated memory, they also copy the effective type from the source, to the destination.

Consider the following:

```
int i = 42;
int *ip;
float *fp;
fp = malloc(sizeof *ip);
ip = (int *)fp;
*fp = 3.14;
memcpy(ip, &i, sizeof(int));
x = *ip;
```

Here the allocated memory first is given the effective type of float, by assigning a float value to it. Then we overwrite the memory again using a memcpy. Finally we dereference the memory as an integer. For this to be legal the memory needs to have the effective type of int. Therefore memcpy (and memmove) does not just copy the memory, they also copy the effective type.

Now consider this:

```
int i = 42;
float f;
memcpy(&f, &i, sizeof(int));
x = f;
```

In this example f has a declared type, and therefore its effective type is always float. Overwriting it using memcpy, is legal, but does not change its effective type and therefore reading f as a float is legal. Now consider this:

```c
int i = 42;
float *fp;
fp = mallof(sizeof(float));
memcpy(fp, &i, sizeof(int));
x = *fp;
free(fp);
```

This superficially looks like the same, as the previous example, only we are now using allocated memory. But this code is UB, because the memory allocated will be assigned the effective type of int by memcopy and dereferencing it as a float is then UB.

character types

Any object can be read and written as a character type irregardless of its effective type. This means that this is legal:

```c
int i = 42;
char *p;
p = (char *)&i;
x = *p;
```

In this case "i" has effective type int, and accessing it as a character type (char) is legal.

The special case for memcpy and memmove also applies to copying characters. consider:

```c
for(i = 0; i < length; i++)
    ((char *)p)[i] = ((char *)x)[i];
```

If we assume that p is pointing to allocated memory, this operation will assign the values and effective type of the memory pointed to by x.

Now consider:

```c
for(i = 0; i < length; i++)
    ((char *)p)[i] = ((char *)x)[i] + 0;
```

Now we are no longer just copying the bytes. This means that the special case no longer applies and the memory pointed to by p will have the effective type of char, not the effective type of whatever x points to.

One curiosity about the exception of character types, is that it is for character types, not for bytes. This means that a char is not guaranteed to be a uint_8. Also somewhat counter-intuitively wchar_t is not a character type.

Unions

Any access of memory using a union, where the union includes the effective type of the memory is legal. Consider:

```
union {
  int i;
  float f;
} *u;
float f = 3.14;
u = &f;
x = u->i;
```

In this case the memory pointed to by "u" has the declared effective type of int, and given that "u" is a union that contains int, the access using the "i" member is legal. It's noteworthy in this that the "f" member of the union is never used, but only there to satisfy the requirement of having a member with a type compatible with the effective type.

This means that when accessing a memory with unknown effective type this can always be done with a "union condom" that contains all types:

```
int function(void *p)
{
  union{ // C99 types only
    bool b;
    char c;
    short s;
    int i;
    long l;
    long long ll;
    float f;
    double d;
  } *uber_union;
  u = p;
  return uber_union->i;
}
```

Non-exceptions

While memcpy, memmove and character types have special properties to accommodate the effective type system, many other functions do not.

The range of functions that would have issues in the standard library includes functions like, calloc, memset, and fwrite. How do these functions impact the effective type of the memory they operate on? There are two possible readings of the standard here.

Reading one

The first reading is to assume that the following line in the standard apply:

"If a value is stored into an object having no declared type through an lvalue having a type that is not a character type, then the type of the lvalue becomes

the effective type"

This would indicate that if a function writes to memory using a character type, the effective type is not impacted. It assumes that functions like calloc, fwrite and memset all use characters, and therefore do not impact the memory's effective type. I think this is the right way to interpret the standard. It does, however, create some issues. Consider:

```
float *fp, *ip;
fp = malloc(sizeof *fp);
*fp = 3.14;

ip = (int *)fp;
memsef(ip, 0, sizeof(*ip));
x = *ip;
```

In this example the allocated memory is given the effective type of float, and then the memory is re-used as memory to store an int. We then re initialize the memory using memset, but memset does not change the effective type, so it still has the effective type of float. Therefor accessing this memory using the type int is UB. One can imagine a lot of similar situations where for instance memory buffers are being reused, and used with fread.

Reading two

There is a different way to interpret the standard.

"If a value is copied into an object having no declared type using memcpy or memmove, or is copied as an array of character type, then the effective type of the modified object for that access and for subsequent accesses that do not modify the value is the effective type of the object from which the value is copied, if it has one."

One could argue that functions like memset and fwrite copy characters, and that the memory being copied to would then inherit the effective type of the memory being copied, if it wasn't declared. Given that the spec, by omitting the effective type of the memory being copied, makes the effective type undefined, any access that isn't made using a character type or a union containing all types is UB. To reiterate, I think this is a possible reading of the standard but I do not think it's the intended reading of the standard.

Effective Types of structs and unions.

One question that has remained unanswered is how effective type deals with structs and unions. We have discussed that unions can be used to access memory with other types than its effective types, but when we use a union or struct to write to a non-declared memory, what effective type does the memory get? Does the memory get the Effective Type of the member or the struct/union?

The standard says nothing about this, but it is obviously pretty important to know. There simply isn't an answer to this that you can find in the spec. But we can make some educated inferences and come to some logical conclusions. Again, I want to make it clear that this is my reading of the standard, and other people may have different readings.

Let's start with struct members:

A member of a structure could conceivably have the effective type of the member, the struct or both at the same time.

Consider:

```
struct {
 int i;
} s;

s.i = 5;
ip = &s.i;
x = *ip;
```

I do not think this was intended to be UB. You can access members of a struct using pointers. If this wasn't possible, then a vast majority of C code would not function. We can clearly infer that it is the intention that this should be possible in C. That means that it has to either have the effective type of int or int and the struct. Given that the spec never says that memory can have more then one effective type, I think just int is more likely. So the answer should be: A struct has the effective type of its members.

So one would assume unions are the same? Let's have a look. Consider:

```
union {
 int i;
 float f;
} *u;

u->i = 5;
x = u->f;
```

This should also be legal. Given that a value can be accessed using a union that "contains" the effective type, this would be legal if the active type was int. It would however also work if the effective type was union, because we are accessing it using the same type. If we instead consider:

```
union {
 int i;
 float f;
} *u;

u->i = 5;
```

```
fp = &u->f;
x = *fp;
```

I think there is some consensus that this is UB, and I'm pretty sure I have seen examples of code like this breaking, in real world compilers. *fp has to access a compatible type, and neither int or the union is a compatible type. If we then consider:

```
union {
 int i;
 float f;
} *u;

u->i = 5;
ip = &u->i;
x = *ip;
```

This I think should work (but I would highly discourage anyone depending on it). This would indicate that the effective type is in fact, int.

So my conclusion is: struct and union members have the effective type of the member.

What if I'm wrong about Effective Types of structs and unions?

The above is just my own personal conjecture of how I think it is reasonable to interpret the standard. Reasonable people may disagree, so let's explore the ramifications of reading the standard differently. The effective type rules doesn't mentions structs. So its reasonable to think that structs and unions are their own effective type, that are different from the types making up those structures and unions. If I was to interpret the standard text only, without any consideration of if it makes sense, or if it could reasonably be the intention of the authors of the standard, then I would probably conclude that this is the correct way to interpret the standard.

This would mean that:

```
struct {
 int i;
} s;

s.i = 5;
ip = &s.i;
x = *ip;
```

is UB. The underlying memory storing `s`, would have effective type of a specific struct, and therefore accessing it as an integer would be illegal. This would obviously break a lot of code, because it would be impossible to take a pointer

off a member of a struct (or union), and then use that pointer to access the struct's content. It would also mean that the following is UB:

```
struct {
 int i;
} s;

struct {
 int i;
 float f;
} *u;

s.i = 5;
u = &s.i;
x = *u.i;
```

Even though we are accessing an int value, that was assigned using an int, with a union that contains an int, and we are accessing it using the int member of that union, this is UB. The memory's effective type is that of the struct, and since the union doesn't contain a struct compatible with that struct, the access is illegal. If this is the correct reading of the standard, then the notion of a "union condom" that contains all possible types, is impossible because it would have to contain all possible unions and structs, and that is near infinite.

Final thoughts, and recommendations.

If you read this document, you may find to your horror an awful lot of C code, probably code you have written, is UB and therefore broken. However just because something is technically UB doesn't mean compilers will take advantage of that and try to break your code. Most compilers want to compile your code and not try to break it. Given that almost no one understands these rules, compilers give programmers a lot of leeway. A lot of code that technically breaks these rules will in reality never cause a problem, because any compiler crazy enough to assume all code is always in 100% compliance with these rules would essentially be deemed broken by its users. If you are using fwrite to fill out a structure, you are just fine. No reasonable compiler would ever break that code.

The issue is not that implementations don't give users leeway, the issue is that it's unclear how much leeway is given.

Many compilers (Like MS Visual Studio) just flat out ignore these rules and will let you break them any way you want. Other compilers like GCC and llvm offer options like no-strict-aliasing that turn off any optimizations relying on these rules. Many projects (most notably the Linux kernel), and security guidelines mandate no-strict-aliasing, in order to get around the issue entirely. If you are unsure or want to be safe, I recommend using these options, especially if you are working on larger team with diverse levels of experience.

If you are unsure or want to be safe, these are my recommendations: (To be clear these are MY recommendations. The standard is much more complex, and will technically let you break some of these rules. These are my simple-ish rules when you want to be sure you are conforming to implementations, and stay at a safe distance from ambiguities, different interpretations and compiler bugs.)

- Don't ever use variables as "memory buffers" that can be written to, outside of memcpy and memmove. Don't ever access a declared variable with an other type then its declared type.
- Don't ever access uninitialized memory for any reason. (reading uninitialized memory is often UB, and can have very surprising results. For instance reading the same uninitialized values twice may yield different results, or branches of your code that read uninitialized memory may be deleted entirely. Uninitialized memory is not a source of entropy)
- Don't use calloc or memset for struct or pointer initialization. (This is probably the most controversial advice on this list. This practice creates hard-to-find second use bugs and the bit representation is not portable. (memset(&p 0x0, sizeof(void *)) is not guaranteed to be set to NULL for example), It can at times be advantageous to "pre-prime" memory using memset, and then initialize the memory for performance)
- Don't ever take a pointer off a member of a union. (Implementations have been known to break code doing this, either because of compiler bugs or being UB.)
- If you take a pointer to a member of a struct, never use that pointer to access anything outside of that member (like other members of the struct)
- Any time you need to type prune, use a union. (Do not access the same memory using a cast pointer. If you have an int and want to access it as a float, make a union, copy the integer to the union and then read the float from the union. Any reasonable compiler will optimize away the copy.)
- Don't ever convert an integer into a pointer. (This is at best non-portable and often run afoul of provenance rules or compiler interpretations of the same)
- Don't ever read memory with a type that is different then it was written.
- If allocated memory is re-used, make sure it is re initialized using the new types it will be used, before reading from it.
- Assume standard lib functions that write memory do so with the type you will use to read that memory. (This may, in theory, be UB, but guaranteed to work on all platforms)
- When in doubt use a union condom, or memcpy in to a declared type.
- Never create a pointer to an area outside of the size plus one byte of the object. Any pointer outside of this may have wrapped or been clamped.

don't assume you know what your machine does in this case.

- Never use a pointer to a freed object for ANYTHING, including comparing it to other pointers. (If you want to test for ABA bugs, convert the pointer to an integer before freeing it and then convert the new pointer to an other integer, and then compare the two integers.)
- Never convert a pointer to integer for any other reason than the above, and debugging/visualization.
- Pointers to different objects do not relate to each other, Never test where two pointers to different object are in relation to one an other, never compute the offset between two objects and never try to access one object using a pointer originating form another.
- Never use Variable length arrays. They have no way of reporting out-of-memory, and the stack is small. They are inherently untrustworthy. they are effectively UB. (So is recursion, unless you have set a hard limit on the number of recursions that have been properly tested on the target platform, then they become platform dependent)
- Do not EVER think you know when it's OK to break the rules, because you know how your compiler/platform works, if you do, the compiler will find a way to break your code in the most insidious way you can('t) imagine.